UML reflections

François Pennaneac'H, Jean-Marc Jézéquel, Jacques Malenfant, Gerson Sunyé

UML Reflections

François Pennaneac'h, Jean-Marc Jézéquel,
Jacques Malenfant, and Gerson Sunyé

IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
email: pennanea,jezequel,malenfant,sunye@irisa.fr

Abstract The UML shares with reflective architectures the idea that self-definition of languages and systems is a key principle for building and maintaining complex systems. The UML is now defined by a four-layer metalevel structure, enabling a flexible and extensible definition of models by metamodels, and even a self-description of the meta-metamodel (the MOF). This metalevel dimension of UML is currently restricted to structural reflection. But recently a new extension to the UML, called the *Action Semantics* (AS), has been proposed for standardization to the OMG. This paper explores how this proposed extension brings a behavioural reflection dimension to the UML. Indeed, we show that it is not only possible but quite effective to use the AS for manipulating UML models (including the AS metamodel). Besides elegant conceptual achievements, such as a metacircular definition of the AS, reflective modeling with the AS leverages on the UML metalevel architecture to provide the benefits of a reflective approach, in terms of separation of concerns, within a mainstream industrial context. A complete model can now be built as an ideal model representing the core concepts in the application, to which non-functional requirements are integrated as fully traceable transformations over this ideal model. For example, this approach paves the way for powerful UML-defined semantics-based model transformations such as refactoring, aspect weaving, application of design patterns or round-trip engineering.

1 Introduction

The UML (Unified Modeling Language) shares with reflective architectures the idea that self-definition of languages and systems is a key principle for building and maintaining complex systems. In its recent versions, the UML adopted a four-layer metalevel structure, which has enabled a flexible and extensible definition of models through metamodels, and even a self-description of the meta-metamodel. We concur with Bézivin and Lemesle [1] that this metalevel architecture is both a sign that models are now becoming first-class entities, but also that modeling now challenges the role of programming as the central activity in software development. But this metalevel dimension of the UML is currently restricted to structural reflection, a restriction that, in our view, holds up this mutation. Fortunately, new additions to the standard are now introducing behavioural reflection capabilities that will, again in our opinion, play a major role in this software development shift.

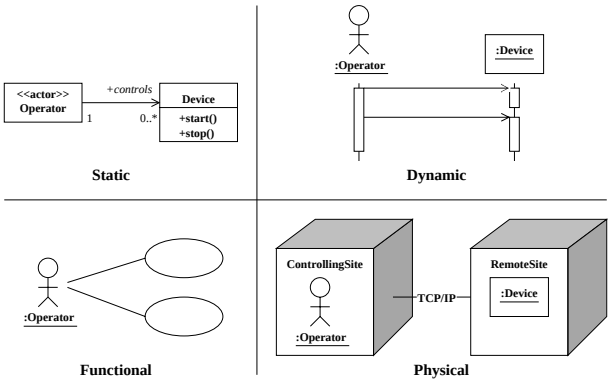

Figure 1. Uml Dimensions

Until recently, the UML lacked precise and formal foundations for several constructs such as transition guard expressions or method bodies, for which it resorted to semantic loopholes in the form of "uninterpreted" strings. Some tools use general purpose programming languages or specific proprietary languages, but yet this is a rather ad hoc and non-standard answer to the problem. Because this was hampering the acceptance of the UML in some industrial circles (e.g., the aerospace or telecom industry), where the strong emphasis on the criticality of software is coped with using precise behaviour description languages and formal verification tools, the Action Semantics (AS) [10] has been proposed for standardization to the Object Management Group (OMG). The AS aims at filling this gap by providing means to annotate UML models with action statements as well as a model of execution for these statements. The AS is introduced as a metamodel integrated into the UML metamodel. Building over what already exists in the CCITT Specification and Description Language (SDL) community [6,11], the integration of the Action Semantics into the UML standard should promote interoperability among tools, and allow for executable modeling and simulation, as well as full code or test cases generation.

This paper explores how this proposed extension brings a behavioural reflection dimension to the UML. While the purpose of the AS is to annotate models with executable specifications for applications source code, we show in this paper that it is not only possible but quite effective to use the AS for manipulating UML models, including even the AS metamodel. Besides providing grounds for neat conceptual achievements, such as a metacircular minimal definition of the AS itself, reflective programming with the AS leverages on the UML metalevel architecture (the UML metamodel is itself an UML model) to provide the benefits of a reflective approach within a mainstream industrial context. Indeed, a model can now be built as an ideal model representing the core concepts in the application, to which non-functional requirements are integrated as transformations over this ideal model.

This novel approach to metamodeling builds on the strong commitment towards separation of concerns that UML also shares with approaches such as AOP (Aspect Oriented Programming) and SOP (Subject Oriented Programming). With its nine views which are like projections of a whole multi-dimensional system onto separate plans, some of them being orthogonal, the UML indeed provides the designer with an interesting separation of concerns, covering four main dimensions of software modeling (see Fig. 1): functional (use cases diagrams express the requirements), static (class diagrams), dynamic (statecharts, sequence diagrams for the specification of behavioural aspects) and physical (implementation diagrams). By applying the AS reflectively to models and metamodels, we have been able to show how designers can carry on, within the UML notational context, activities such as behaviour-preserving transformations [23] (see § 4.2), design pattern application [12] (§ 4.3) and design aspects weaving [15] (§ 4.4).

The rest of the paper is structured as follows. Section 2 recalls the principles of the UML metalevel architecture ; besides explaining the architecture itself, this section also discusses the parallel with well-known object-oriented programming languages metalevel architectures. Section 3 reviews the Action Semantics as it is currently submitted for standardization at the OMG and it explains how it can be used for reflective modeling. Section 4 shows the interest of using the Action Semantics at the metamodel level for specifying and programming model transformations in several contexts. Related work are discussed in Section 5, and a conclusion summarizes our contributions.

2 The UML with the Action Semantics

2.1 From modeling to metamodeling

The UML is all about defining models for executable software artifacts, and more precisely objects. Commonly, the UML is used to define models for objects that will eventually execute on some computer to carry on a computation. A model usually abstracts implementation details of executable objects into entities, which are depicted using a graphical notation and, more and more importantly, a standard XMI serialization format. On the other hand, a model can provide more information that a crude program, mainly by representing and by giving details about relationships between objects, which are implemented by only a few expressible relationships in programming languages (client, inheritance, sometimes agregation and a few others).

If models can describe executable objects and their relationships, it has been soon recognized that they could as well describe other artifacts, and more precisely models. After all, models are built of entities and relationships. Hence, models of models, or metamodels, can describe what kinds of entities and relationships are found in models. Without surprize, metamodels being themselves models, the idea of describing them using metametamodels comes immediately to minds. To avoid a potential infinite metaregression, a fixed-point must be sought to define precisely and completely a modeling architecture.

2.2 A standard four-layer metalevel architecture

The UML follows a four-layer modeling architecture, each layer being a model of the layer under, the last one being a model of itself. The first layer, called M_0, holds the executable ("living") entities when the code generated from the model is executed, i.e. running objects, with their attribute values and links to other objects. The second layer M_1 is the modeling layer. It represents the model as the designer conceives it. In UML, this is the place where well-known classes, associations, state machines,... are defined (via the nine views, quoted above). The running objects are "instances" of the classes defined at this level. The third level M_2 is the metamodel level, i.e. a description of what a syntactically correct model is. Finally the fourth level M_3 is the meta-metamodel level, i.e. the definition of the metamodel syntax, such as the syntax of the UML metamodel. UML creators chose a four-layer architecture because it provides a basis for aligning the UML with other standards based on a similar infrastructure, such as the widely used Meta-Object Facility (MOF).

This four-level architecture brings a form of structural reflection to the UML. Each level defines the syntax and semantic constraints of models one level below. Defining UML as a metamodel allows for easy extension of UML to serve specific modeling needs, while the MOF serves as a unique root to define several modeling languages, perhaps adapted to specific application areas. Bézivin and Lemesle have explored the underpinnings and consequences of this metalevel architecture and predict a mainstream role to modeling in future software development processes [1].

In terms of structural reflection, however, the UML architecture is somewhat restricted. As the major restriction, notice that the relationship between levels is neither explicitly defined nor explicitly used in the definition of UML [1]. This relationship is not precisely instantiation, since it is a many-to-many relationship, many elements of one level usually concur to the definition of one or more elements one level below. Moreover, a more traditional instantiation relationship can be depicted between elements of adjacent levels, such as between classes defined at level M_1 and the objects at level M_0. Bézivin and Lemesle suggest the term "based-on" to name this relationship between levels, a convention we adopt here. Again, we concur with these authors that the two relationships, "based-on" and "instance-of", will have to be reified into UML and the MOF for this metalevel architecture to fully bear fruits.

2.3 Comparisons with other reflective architectures

Unsurprisingly, the metalevel architecture of UML is reminescent of several well-known architectures in reflective languages and systems. Loops [3] has a metaclass/class/instance architecture where the metaclass `Metaclass` is the root of the instantiation hierarchy, playing a role similar to the one of the MOF. The metaclass `Class` defines the structure and behaviour of a Loops class and therefore plays a role similar to the one of the UML metamodel, which defines the syntax of a UML model.

The question of whether the UML four-level architecture can collapse into three levels, in a similar way as Briot and Cointe did for Loops with the ObjVLisp architecture [4,5,9] comes immediately to the mind; we will return to this next. Maintaining a four-level architecture though has the strong advantage of allowing many different metamodels that still share the same basis for their definition (see [1]).

The fact that many different model elements concur to define one particular object is also reminescent of metalevel architectures for concurrent object-oriented languages such as CodA [21], among others. The organization of these model elements in nine views is similar to the multi-model reflection framework of AL-1/D [22], which has been adopted by many object-oriented reflective middleware, such as OpenORB [2], and operating systems, such as μChoices [27].

2.4 Metamodeling in practice

As models are becoming growingly complex, the use of sophisticated tools to build and manipulate them is now mandatory in industrial contexts. Thanks to the metalevel architecture of UML, more and more of these tools are built around this reflective representation of models and metamodels. For practical reasons though, full-fledged four-level tools are still rare. One of the potential simplification when the tools choose to specialize themselves into UML modeling is to collapse the level M_3 onto the level M_2, as we have pointed out before.

Although there is no strict one-to-one mapping between all of the MOF meta-metamodel elements (M_3) and the UML metamodel elements, the two models are interoperable. In fact, the UML core package metamodel and the MOF are structurally quite similar. A reason explaining this similarity is that self-described UML metamodels have been proposed before the standardization of the MOF. Adding the obvious influence of UML on the design of the MOF explains a relatively tight compatibility between a subset of the UML metamodel and a subset of the MOF. This design implies that the UML metamodel (a set of class diagrams) is itself a UML model (pretty much in the same way that the metaclass `Class` in ObjVlisp is self-describing and therefore an instance of itself). The Fig. 2 shows this inclusion. The chief interest of this property, as tools are concerned, is the fact that tools built to manipulate UML models can very well, and indeed do apply equally well to metamodels, a specificity we heavily rely on for our study, as we will see shortly.

3 The Action Semantics and its reflective properties

3.1 Motivations for an Action Semantics in UML

Traditional modeling methods which do not have support for any action language to specify the behaviour of model elements have focused on separating analysis and design, i.e. the *what the system has to do* and the *how that will be achieved*. If this separation clearly has some benefits, such as allowing the

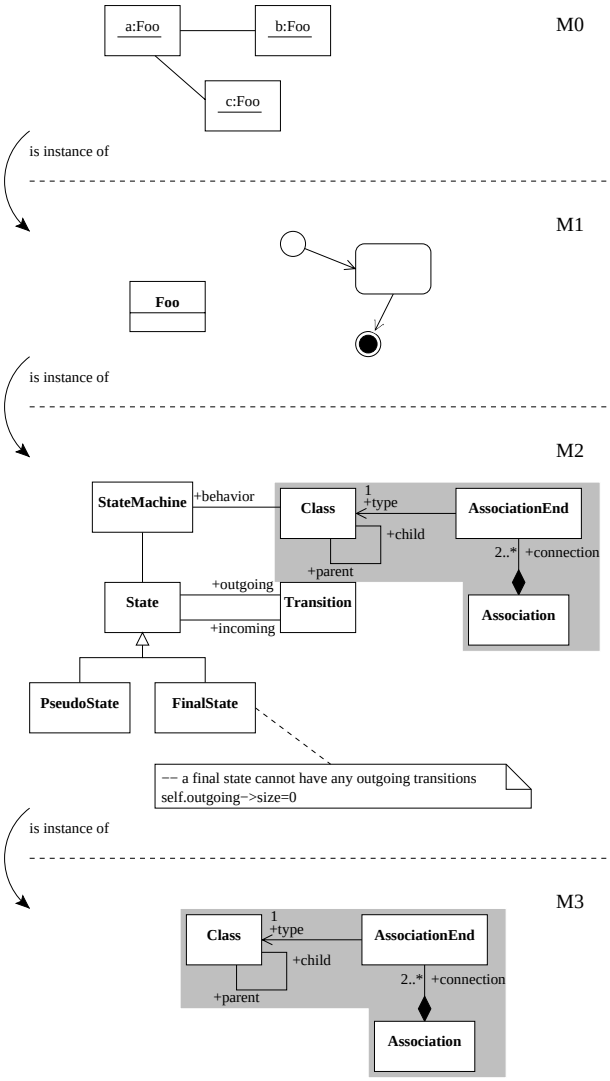

Figure 2. UML layers (partial and simplified)

designer to focus on system requirements without spreading itself too thin with implementation details, or allowing the reuse of the same analysis model for different implementations, it also has numerous drawbacks. The main one is that this distinction is a hard job in practice, not to say impossible: the boundaries are vague; there is no criteria for deciding what is analysis, and what is not. Rejecting some aspects from analysis makes it incomplete and imprecise; trying to complete it often force to mix some *how* issues for describing the most complex behaviour.

Therefore, three reasons essentially drive the inclusion of an Action Semantics into UML:

1. First and fundamental, the incompleteness in models due to the inability to state often crucial points about the behaviour of model elements becomes a major shortcoming in many contexts.
2. Second, a consequence of the first, the inability to check models early in the software development process, something common now in some critical-software industries using the SDL and related tools, is hampering the acceptance of UML outside the traditional circles of business-management applications.
3. Third, as mentioned earlier, the UML pushes towards a shift of software development efforts from programming to modeling, and a crucial technology to enable this shift is code generation directly from models, which again needs a complete specification of behaviour into models.

UML is quite crude at expressing behaviour. To date, no really defined element in the UML could specifically provide means for such expression. UML allows user to include expressions, but only in the form of so-called "uninterpreted" strings, for which no semantics is given. Interpretation is left to the user. This is true for example in state diagrams, where the specification of a guard on a transition is conceptually realized by a boolean expression, but nothing within UML metamodel allows the designer to express such a condition on the guard and check for at least this basic semantic constraint. This lack of more formal foundations for important behavioural aspects of models is experienced by more and more users.

Also annoying is the fact that models are neither executable nor checkable, simply because they are incompletely specified in the UML. This makes it impossible to verify and test early in the development process, something common now in some critical-software industries using the SDL and related tools. Model checking techniques, for example, would be inapplicable to UML models if elements as crucial as guards on transition cannot be expressed with a precise semantics.

Finally, code generation from models is a feature of many commercial tools today. Although the generation of code templates to be filled by programmers already boosts productivity, few projects never come back to models during the implementation. When they do, keeping models and source code in synchronization is more and more a burden. Therefore, the ideal would be to work out the application by modeling from end to end. That is, the model should be defined

with enough precision, including behaviour, to enable the generation of the entire application source code, therefore collapsing all problems due to the dual evolution of models and source code. Maybe less important, but increasingly present in tools, the ability to simulate, that is to execute, the model prior programming also requires completely specified models.

Faced with these shortcomings, tool providers came with several ad hoc solutions, such as adopting a particular programming language to write behavioural expressions in otherwise "uninterpreted" strings (e.g., Java or C++). This kind of ad hoc solutions merely and inevitably trade incompleteness for inconsistencies in UML models when different languages are adopted in different models. Indeed, the interpretation is delegated to the modeling tool, therefore breaking the interoperability of models, a hardly gained property through years of long and painful unification and standardization processes.

The Action Semantics proposal aims at providing modelers with a complete, software-independent specification for actions in their models. The goal is to be able to use UML as an executable modeling language [13], i.e. to allow designers to test and verify early and to generate 100% of the code if desired. It builds on the foundations of existing industrial practices such as SDL, Kennedy Carter's [16] or BridgePoint [24] action languages[1]. But contrary to its predecessors, which all were proprietary, the Action Semantics aims at becoming an OMG standard, a common base and formalism for all the existing and future action languages (mappings for existing languages to Action Semantics are proposed).

3.2 A quick reference to the AS

The Action Semantics proposal is based upon three abstractions:

- A metamodel: it extends the actual UML1.4 metamodel, augmenting the uninterpreted items with a precise syntax of actions (see Fig 3). New subclasses of metaclass *Action* are introduced to cope with the usual programming constructs : primitives action such as the creation, deletion or modification of objects, creation or deletion of links between objects, control structures or the sending of a message are defined. Mechanisms for grouping actions (into procedures, for instance) while specifying their partial order of execution are provided. The Action Semantics proposal for the UML does not enforce any notation (i.e. surface language) for the specification of actions. This is intentional, as the Action Semantics goal is certainly not to define a new notation, or to force the use of a particular existing one. But the Action Semantics was conceived to allow an easy mapping of classical languages such as Java, C++ or SDL[2]. Thus, designers can keep with their favorite language, without any learning overhead for a new one.
- A model of execution: it is the UML model of a virtual machine for executing UML specifications. It defines abstractions for modeling the evolutions of the

[1] all the major vendors providing an action language are in the list of submitters.
[2] Some of these mappings are illustrated in the AS specification document [10].

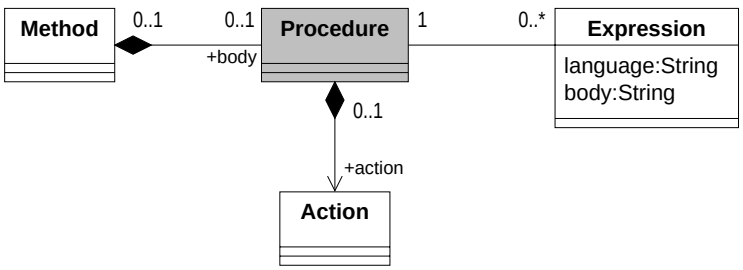

Figure 3. Action Semantics immersion into the UML

objects in a system at runtime. The life of an object is modeled by its *history*, i.e. a sequence of *snapshots*. Each snapshot shows the value of attributes and links to other objects, and is immutable. Each change to an object, either to one of its attribute values or links, yields a new snapshot in the history.
– A semantics of actions: the execution of an action is precisely defined with a *life-cycle* which unambiguously states the effect of executing the action on an instance of the execution model. Every life-cycle is made of basic steps of execution called *productions*. A production has a precondition and a postcondition. The precondition is evaluated with respect to the current snapshot. When it is true, execution proceeds to the next step in the life-cycle and a new snapshot validating the properties stated in the postcondition is computed.

3.3 Making the AS reflective

The AS was originally conceived for precisely specifying the behaviour of models. We advocate the extension of its scope beyond this basic role. An UML execution engine, i.e. an implementation of the AS model of execution is originally dedicated to the manipulation of M_0 instances of UML models. Such manipulations are specified at the M_1 level, as part of the whole model of the application. But since both (1) the UML meta-model and (2) the UML execution model for the AS are themselves UML models, we can use the AS to specify the evolution of these models:

– In the first case, thanks to the four-level architecture of the UML, an AS specification would manipulate instances of M_2 level, i.e. UML models. Then, an AS specification describes a *model transformation* (meta-programming).
– In the second case, an AS specification would manipulate instances of the execution model, i.e. the objects at runtime (a representation of M_0 level called a snapshot). Then, the AS specification describes the transformation from one snapshot to the resulting one, that is the *semantics of the AS itself* (reflexivity applied to the execution engine specification).

We propose to combine these two approaches into reflexive meta-modeling tools: the same execution engine would then apply to both the execution of metamodel transformations and the execution of models.

Before detailling the use of the AS for metamodel transformations in the next section, let us now explore the interests of a reflexive definition for the AS. First it would help to structure the current AS definition document (more than 250 pages long) around a kernel of reflexively defined primitive actions and thus minimize the amount of work needed for implementations. These primitive actions fall in the following categories:

- reading and writing of attributes
- creation and deletion of objects
- creation, reading and deletion of links between objects
- communication among objects (call action, signal, exceptions)

These primitive actions are defined with production rules specifying the transition between two snapshots, which are themselves M_0 level UML models, hence the meta-circular definition. Other AS actions could then be defined operationaly based on this kernel of primitive actions. For instance, *SynchronousInvocation-*

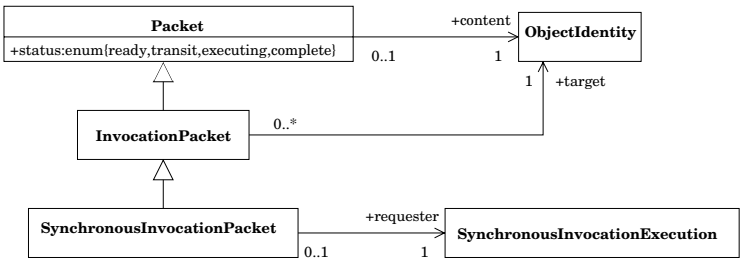

Figure 4. SynchronousInvocationAction model of execution

Action is defined as an action that sends a message to a target object and blocks until it receives an answer. The semantics of execution for this action is (partially) described in Fig. 4 and completed with the following production extracted from the specification:

```
Production 1 : synchronous invocation action generates a synchronous
invocation packet and blocks itself.
precondition: self.status = #ready
postcondition: self.status@post = #executing -- put call in executing
-- state while subordinate execution proceeds.
packet:SynchronousInvocationPacket.isNew and -- create a call packet
packet.requester@post=self and
packet.status@post= #ready and
packet.target@post=self.input_target() and
packet.content@post=self.input_argument
```

It is obvious to deduce a sequence of AS primitive actions which creates and initializes a *packet* conforming to this postcondition.

```
packet := new SynchronousInvocationPacket -- a CreateObjectAction
packet.requester := self -- a CreateLinkAction
packet.status := #ready -- a WriteAttributeAction
packet.target := self.target -- a ReadLinkAction then a CreateLinkAction
packet.content := self.argument -- n ReadLinkActions, n CreateLinkActions
```

Until now, this has been left out of the OMG specification, where productions focus on giving a precise meaning to the execution of actions, putting aside the building of compliant and efficient execution engines. Productions do not explicitly state *how* to compute the next snapshot in an object history and thus are an inappropriate formalism for tools implementors. This idea could also be used to let the user define new actions based on primitive ones, thus extending the AS.

Finally, we propose to give the programmer the full power of UML reflexivity by providing M_0 level entities with a link to the M_1 level, and M_1 level entities a link to M_2 level in such a way that she can freely navigate between the various levels of modelisations, both at runtime and at design time.

4 Leveraging the UML Reflexivity

In this section, we discuss why and show how to use the behavioural reflection introduced by the Action Semantics to implement model transformations. We present three different uses for this approach: implementing refactorings, applying design patterns and weaving design aspects.

4.1 Separating concerns: codesign

Using the UML meta-modeling architecture and the Action Semantics for specifying transformations is appealing: the development of meta-tools capitalizes on experience designers have gained when modeling UML applications. Some recurrent problems suddently vanish: portability of transformations is ensured for all UML-compliant tools with access to the metamodel, there is no learning-curve for the writing of new meta-tools, as it is pure UML and any development process supporting the UML applies to the building and reuse of transformations [18]. This paves the way to off-the-shelf transformation components.

We strongly believe that this use of the Action Semantics will change the traditional software development process. More concretely, the Action Semantics is an important step towards the use of UML in an effective development environment, since it offers the possibility of enacting early design models and to evolve or refine it until its implementation. The development approach we propose here starts with an early design model, created by the designers from an analysis model. This model is completely independent of the implementation

environment, it assumes an "ideal world", where the processing power and the memory are infinite, there is no system crash, no transmission error, no database conflicts, etc. Since this model encloses Action Semantics statements, it can be enacted by the Action Semantics machine and validated. Once the validation is finished, the designer can add some environment-specific aspects to the design model (database access, distribution), apply design patterns and restructure the model using design refactorings.

As already outlined, transformations rank in two categories: the ones related to the application domain and those involved in generating efficient implementations for the target platform. The following example illustrates the difference: if the designer knows a collection of objects has to be notified when another object changes, then she annotates the corresponding classes as collaborating into an Observer pattern [12]. A generic transformation supporting this pattern adds an `update` method to every observer. Specific transformations for implementing the pattern offer designers choices that fit implementation trade-offs: execution speed vs. memory footprint, point-to-point notification vs. broadcasting, depending on requirements on the underlying hardware. This last transformation is not at all related to the application, and must not distract the designer from its application refinement.

The two categories are not exclusive : some transformations bridge the application domain and the implementation domain, thus falling into both categories. These transformations perform the "weaving" of the two aspects into a single implementation model.

4.2 Design Refactorings

Refactorings [23] are behaviour-preserving transformations, used to improve the design of object-oriented programs. We believe that refactorings are an important artifact to software development, and we are interested in bringing them to the design level by means of a UML tool. The implementation of refactorings in UML is an interesting task, since *design* refactorings – as opposed to *code* refactorings – should work with several modelling elements shared among several views of a model. This is also a challenge, since some refactorings (namely moving features) are difficult to implement since we must take into account different UML elements, such as OCL constraints and state charts.

The Action Semantics represents a real gain for refactoring implementation, not merely because it directly manipulates UML elements, but also because of the possibility of combining it with OCL rules to write pre and post-conditions. More precisely, as refactorings must preserve the behaviour of the modified application, they cannot be widely applied. Thus, every refactoring ought to verify a set of conditions before the transformation is carried out.

Below we present an example of a simple refactoring, the generalization of equivalent Attributes. In the UML metamodel, an Attribute belongs to a Class, its *Owner*. It may have *Sisters*, the children of its owner. In addition to the equivalence, which must be satisfied for exactly one Attribute of each sister, two other preconditions should be satisfied. First, private Attributes can not be

moved, since they are not visible outside the scope of the owner and are not inherited. Second, the owner must have exactly one parent. These conditions and the transformation itself are defined in OCL and in Action Semantics as follows[3]:

```
Attribute:: generalize
pre:
    self . visibility  <>private and
    self .owner.parent.size = 1 and
    self .owner.parent.child→forAll(aClass:Class|
       aClass.feature→select(anAttr|
          anAttr.oclIsKindOf(Attribute))→exists(a|
             a.isBasicEquivalentTo(self )))
actions:
    let newAttribute := self.copy()
    self .owner.parent.addFeature(newAttribute)
    self .owner.parent.child→forAll(aClass:Class|
       aClass.feature→select(anAttr|
          anAttr.oclIsKindOf(Attribute) and
          anAtrr.isBasicEquivalentTo(self))→forAll(each | each→delete()
post:
    self@pre.owner.parent.features→exists(a:Attribute|
       a.isBasicEquivalentTo(self ))) and
    not self@pre.owner.parent.child→forAll(aClass:Class|
       aClass.feature→select(anAttr:Attribute|
          anAttr.oclIsKindOf(Attribute))→exists(a:Attribute|
             a.isBasicEquivalentTo(self )))
```

An OCL expert might rightfully notice that the operation *child* is not defined neither in the OCL documentation nor as an additional operation in the UML metamodel. We have defined it symmetrically to the *parent* operation, defined for Classes.

4.3 Design Patterns

Another interesting use for Action Semantics is the application of Design Patterns, i.e. the specification of the proposed terminology and structure of a pattern in a particular context (called instance or occurrence of a pattern). In other words, we envisage the application of a pattern as a sequence of transformations that are applied to an initial situation in order to reach a final situation, an explicit occurrence of a pattern.

This approach is not, and does not intend to be, universal since only a few patterns mention an existing situation to which they could be applied (see [7] for further discussion on this topic). In fact, our intent is to provide designers with metaprogramming facilities, so they are able to define (and apply) their

[3] Since the Action Semantics does not have an official surface language, we adopt an "OCL-based" version in our examples.

own variants of known patterns. The limits of this approach, such as pattern and trade-offs representation in UML, are discussed in [26].

As an example of design pattern application, we present below a transformation operation that applies the Proxy pattern. The main goal of this pattern is to provide a placeholder for another object, called *Real Subject*, to control access to it. It is used, for instance, to defer the cost of creation of an expensive object until it is actually needed:

Class :: addProxy
pre:
    let classnames = self.package.allClasses→**collect**(each : Class | each.**name**) in
    (classnames→**excludes**(self.**name**+'Proxy') **and**
    classnames→**excludes**('Real'+self.**name**))
actions:
    let **name** := self.**name**
    let self.**name** := **name**.**concat**('Proxy')
    let super :=
        self.package.addClass(**name**,self.allSuperTypes(),{}→**including**(self))
    let real :=
        self.package.addClass('Real'.**concat**(**name**),{}→**including**(super),{})
    let ass := self.addAssociationTo('realSubject',real)
    self.**operations**→**forAll**(op : Operation | op.moveTo(real))

This operation uses three others (actually, refactorings), that will not be precisely described here. They are however somewhat similar to the *remove-Class()* operation presented above. The first operation, *addClass()*, adds a new class to a package, and inserts it between a set of super-classes and a set of subclasses. The second, *addAssociationTo()*, creates an association between two classes. The third, *moveTo()*, moves a method to another class and creates a "forwarder" method in the original class.

This transformation should be applied to a class that is to play the *role* of real subject [4]. Its application proceeds as follows:

1. Add the 'Proxy' suffix to the class name;
2. Insert a super-class between the class and its super-classes;
3. Create the *real subject* class;
4. Add an association between the *real subject* and the *proxy*
5. Move every method owned by the *proxy* class to the *real subject* and create a forwarder method to it (move methods).

As we have explained before, this is only one of the many implementation variants of the Proxy pattern. This implementation is not complete, since it does not create the *load()* method, which should create the *real subject* when it is requested. However, it can help designers to avoid some implementation burden, particularly when creating forwarder methods.

[4] Patterns are defined in terms of roles, which are played by one or more classes in its occurrences

4.4 Aspect Weaving

Finally, we would like to show how Action Semantics can support the task of developing applications that contain multiple aspects. Aspects (or concerns) [17,28] refer to non-functional requirements that have a global impact on the implementation. The approach used in dealing with this is to separate these aspects from the conceptual design, and to introduce them into the system only during the final coding phase. Ultimately, the merging of aspects should be handled by an automated tool. In our example, we attempt to show how aspects can be weaved as early as the design level through model transformation [14], using the Action Semantics to write the transformation rules.

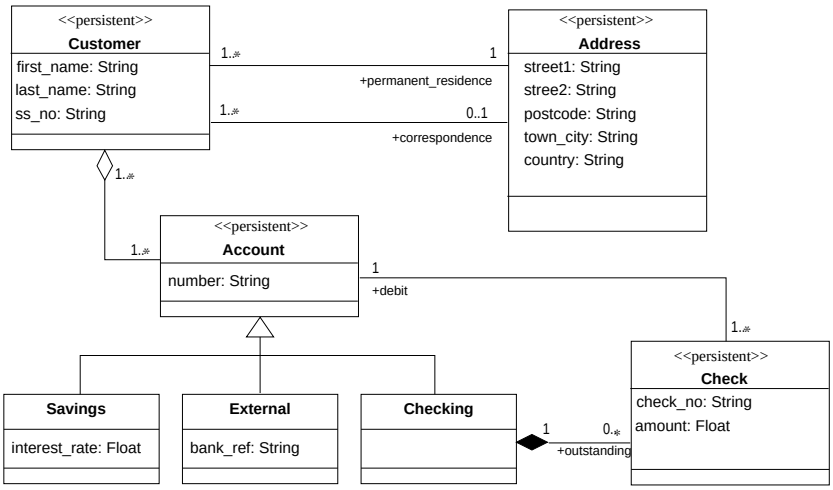

Figure 5. Information Management System for Personal Finance

The class diagram in Fig. 5 illustrates a model of a bank's personal-finances information-management system. In the original system, the accounting information was stored in a relational database and each class marked with the "persistent" stereotype can be related to a given table in the database. The aim of this re-engineering project is to develop a distributed object-oriented version of the user front-end to support new online access for its customers. One of the non-functional requirements is to map these "persistent" objects to the instance data stored in the relational database. The task involves writing a set of proxy classes that hide the database dependency, as well as the database query commands. An example of the required transformation is illustrated by the model in Fig. 6. In this reference template, the instance variable access methods are generated automatically and database specific instructions are embedded to perform the necessary data access.

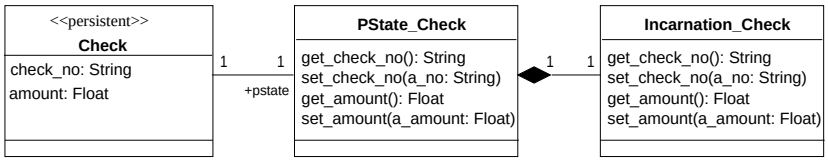

Figure 6. Persistence proxies and access methods

Since the re-engineering is carried out in an incremental manner, there is a problem with concurrent access to the database during write-back commits. The new application must cooperate with older software to ensure data coherence. A provisional solution is to implement a single-ended data coherence check on the new software. This uses a timestamp to test if data has been modified by other external programs. If data has been modified since the last access, all commit operations will be rolled back, thus preserving data coherence without having to modify old software not involved in this incremental rewrite. Fig. 7 shows the template transformation required. It adds a flag to cache the timestamp and access methods will be wrapped by timestamp-checking code.

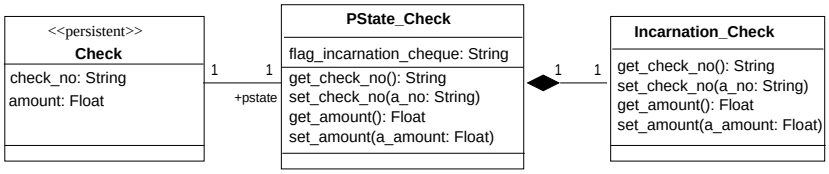

Figure 7. Timestamp cache flag for concurrent data coherence

The metaprogram needed to generate the proxy classes of figures 6 and 7 is composed of several operations. The first one is defined in the context of a Namespace (i.e. the container of UML modeling elements). It selects all classes that are stereotyped 'persistent' and sends them the *implementPersistent()* message:

Namespace:: implementPersistentClasses
actions:

self . allClasses →**select**(each : Class | each.stereotype→**notEmpty**)→
select(each : Class | each.stereotype→**first** . **name** = 'persistent')→
forAll(each : Class | each.implementPersistent)

The *implementPersistent()* operation is defined in the context of a Class. This operation will first create two classes, *state* and *incarnation*, and then creates, in these classes, the access methods to its own stereotyped attributes. This operation is defined as follows:

```
Class :: implementPersistent
actions:
        let pstate :=
                self.package.addClass('PState_'.concat(pclass.name),{},{})
        pstate.addOperation('Load'); pstate.addOperation('Save')
        self.addAssociationTo(pstate, 1, 1)
        let incarnation :=
                self.package.addClass('Incarnation_'.concat(pclass.name),{},{})
        pstate.addCompositeAssociationTo(incarnation, 1, 1)
        let attrs := self.allAttributes→
                select(a : Attribute| a.stereotype→notEmpty)
        attrs→select(a : Attribute | a.stereotype→first.name = 'getset')→
                forAll(a : Attribute |
                        pstate.createSetterTo(a); pstate.createGetterTo(a)
                        incarnation.createSetterTo(a); incarnation.createGetterTo(a))
        attrs→select(a : Attribute | a.stereotype→first.name = 'get')→
                forAll(a : Attribute |
                        incarnation.createGetterTo(a); pstate.createGetterTo(a))
        attrs→select(a : Attribute | a.stereotype→first.name = 'set')→
                forAll(a : Attribute |
                        pstate.createSetterTo(a); incarnation.createSetterTo(a))
```

The creation of the access methods is implemented by the *createSetterTo()* and *createGetterTo()* operations. They are both defined in the Class context and implement a similar operation. They take an Attribute as parameter and create a Method for setting or getting its value. These operations use two other operations, *createMethod()* and *createParameter()*, which are explained above:

```
Class::createSetterTo(att : Attribute)
actions:
        let newMethod := self.createMethod('set_'.concat(att.name))
        newMethod.createParameter('a_'.concat(attrib_name), att.type, 'in')
```

```
Class::createGetterTo(att : Attribute)
actions:
        let newMethod := self.createMethod('get_'.concat(att.name))
        newMethod.createParameter('a_'.concat(attrib_name), att.type, 'out')
```

The *createMethod()* operation is also defined in the Class context. Its role is to create a new Method from a string and to add it to the Class:

```
Class::createMethod(name : String)
actions:
        let newMethod := Method.new
        let newMethod.name := name
        self.addMethod(newMethod)
        let result := newMethod
```

Finally, the *createParameter()* operation creates a new parameter and adds it to a Method, which is the context of this operation:

```
Method::createParameter(name : String, type : Class, direction : String)
actions:
        let newParameter := Parameter.new
        let newParameter.name := name
        newParameter.setType(type)
        newParameter.setDirection(direction)
        self .addParameter(newParameter)
        let result := newParameter
```

The attractiveness of this approach is not immediately evident. Let us consider a different implementation for the persistent proxy of Fig. 6. In the case where there are composite persistent objects, it is possible to use a single persistent state proxy for a composite object and all its components (see Fig. 8). Through the use of metaprogramming, it is now possible to consider these different implementation aspects independently from the implementation of concurrency. It enables the designer to conceptualize the modifications in a manageable manner. Making changes to a model by hand as a result of a change in an implementation decision is not a viable alternative as it is laborious and error-prone.

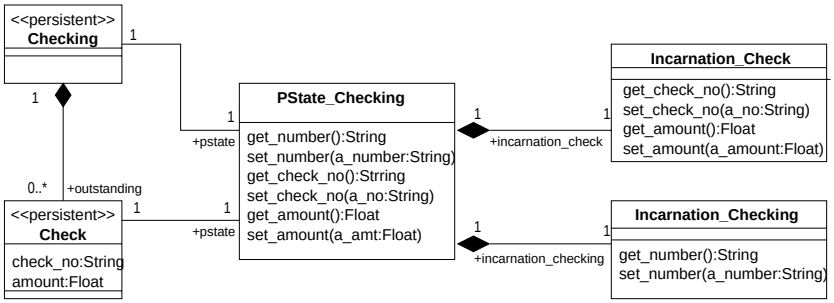

Figure 8. Implementation template for shared proxy

Metaprogramming using the Action Semantics facilitates the revision of implementation decisions by representing them at a higher level of abstraction. It also leverages the execution machine for the Action Semantics by using it to perform the model transformation.

5 Related work

Because the complete UML specification of a model relies on the use of uninterpreted entities (strings), with no well-defined and accepted common formalism and semantics, commercial UML tools often use metaprogramming languages to manipulate models. This is the case, for instance, of Softeam's Objecteering that

uses J (a "Java-like" language [25]), and of Rational Rose or Ilogix' Rhapsody which both use Visual Basic.

In the worst cases – that is for most of the modeling tools – these statements are simply inserted at the right place into the code skeleton. The semantics of execution is then given by the specification of the programming language. Unfortunately, this often implies an over-specification of the problem (for example, in Java, a sequential execution of the statements of a method is imposed). Verification and testing are feasible only when the source code is available, far too late in the development process. Moreover, the designer must have some knowledge of the way the code is generated for the whole model to have a good understanding of the implications of her inserted code (for instance, if you are using Rhapsody and its C++ code generation tools, a knowledge of the way the tool generate code for associations is required for using them in your own code).

At best, the modeling tool has its own action language, and then the model may be executed and simulated in the early development phases, but with the drawbacks of no standardization, no interoperability, and two formalisms for the modeler to learn (the UML and the action language). The purpose of these languages is similar to the one we look for when using the Action Semantics as a metaprogramming language. There are, however, several advantages in favor of a standard Action Semantics. Users of these tools have toneed not learn yet a new language. Thanks to a tight integration into UML, the Action Semantics leverages other UML facilities, such as OCL, and in particular its navigation facilities.

As reflection is concerned, we have already pointed out in Section 2 the similarities and differences between the structural reflection aspects of UML and those of reflective object-oriented languages. On the behavioural reflection side, most of the existing models are based on an interpretive or compilative pattern. In the first case, reflection is introduced in the language by reifying a metacircular interpreter (consider 3-Lisp and other reflective functional languages), while in the second it is introduced by reifying compilation strategies in the compiler either statically (consider OpenC++) or dynamically (consider Smalltalk and OpenJIT). Elements of the run-time systems are also reified into the language.

Our approach to behavioural reflection in UML is rather based on an execution model close to operational semantics where computations are seen as sequences of states and computation steps transform an initial state into a following state in an history. Thanks to the integration with UML, reflection is achieved by having states represented as models and the Action Semantics enabled to manipulate these models. Hence, reflection in the UML with the AS appears closer to the work on rewriting systems [19,8] and rewriting logics [20] than the more traditional interpretive or compilative approaches.

6 Conclusion

With the standardization of the Action Semantics by the OMG, the metalevel architecture of the UML is now supported by a language allowing the speci-

fication of actions in a portable way. In this paper, we have shown how the Action Semantics brings a behavioural reflection dimension to the UML. We have shown that it is not only possible but quite effective to use the AS for manipulating UML models, including the AS metamodel. Behavioural reflection has been brought to the fore by a metacircular definition of the Action Semantics. To enable reflective metamodeling in the AS, we have reified in UML a meta-of link that allows actions defined in a model to act upon model elements through their description at the metamodel level, much in the same way code in a reflective object-oriented language can access class properties through metaclasses.

Applications of reflective modeling with the AS leverages on the UML metalevel architecture to provide the benefits of a reflective approach, in terms of separation of concerns, within a mainstream industrial context. A complete model can now be built as an ideal model representing the core concepts in the application, to which non-functional requirements are integrated as fully traceable transformations over this ideal model. For example, we have illustrated how this approach paves the way for powerful UML-defined semantics-based model transformations such as refactoring, aspect weaving, and application of design patterns.

An implementation conforming to the current version of the Action Semantics specification is in development in UMLAUT [5], a freely available UML modeling tool. The complete integration between the Action Semantics and the UML in Umlaut provides an excellent research platform for the implementation of design patterns, refactorings and aspects.

References

1. J. Bézivin and R. Lemesle. Reflective modelling scheme. In *Electronic Proceedings of the OOPSLA'99 Workshop on Object-Oriented Reflection and Software Engineering, OORaSE'99*, pages 107–122, 1999. web site: http://www.disi.unige.it/person/CazzolaW/OORaSE99.html.
2. G. Blair, G. Coulson, F. Costa, and H. Duran. On the design of reflective middleware platforms. In *Proceedings of the Reflective Middleware Workshop, RM 2000*, 2000.
3. D. Bobrow and M. Stefik. *The Loops Manual*. Xerox PARC, Palo Alto CA, USA, December 1983.
4. J.-P. Briot and P. Cointe. The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language. In *Proceedings of ECAI'86*, pages 225–232, 1986.
5. J.-P. Briot and P. Cointe. A Uniform Model for Object-Oriented Languages Using the Class Abstraction. In *Proceedings of IJCAI'87*, pages 40–43, 1987.
6. CCITT. *Red Book, SDL, Recommendation Z.100 to Z.104*, 1984.
7. M. Cinnide and P. Nixon. A methodology for the automated introduction of design patterns. In *International Conference on Software Maintenance*, Oxford, 1999.
8. M. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. In *Proceedings of the First International Conference on Reflection, Reflection'96*, pages 263–288, 1996.

[5] `http://www.irisa.fr/UMLAUT/`

9. P. Cointe. Metaclasses are first class: the objvlisp model. In *Proceedings of OOPSLA'87*, pages 156–167. ACM, 1987.
10. T. A. S. Consortium. Updated joint initial submission against the action semantics for uml rfp, 2000.
11. J. Floch. Supporting Evolution and Maintenance by using a Flexible Automatic Code Generator. In *Proceedings of the 17th International Conference on Software Engineering*, pages 211–219, Apr. 1995.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
13. O. M. Group. Action semantics for the uml rfp, ad/98-11-01, 1998.
14. W. Ho, F. Pennaneac'h, and N. Plouzeau. Umlaut: A framework for weaving uml-based aspect-oriented designs. In *Technology of object-oriented languages and systems (TOOLS Europe)*, volume 33, pages 324–334. IEEE Computer Society, June 2000.
15. R. Keller and R. Schauer. Design components: Towards software composition at the design level. In *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311. IEEE Computer Society Press, Apr. 1998.
16. Kennedy-Carter. Executable UML (xuml), http://www.kc.com/html/xuml.html.
17. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, N.Y., June 1997.
18. P. Kruchten. *Rational Unified Process: an Introduction*. Addison-Wesley, Reading/MA, 1998.
19. M. Kurihara and A. Ohuchi. An Algebraic Specification and an Object-Oriented Implementation of a Reflective Language. In *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architectures*, pages 137–142, November 1992.
20. J. Malenfant, C. Dony, and P. Cointe. A Semantics of Introspection in a Reflective Prototype-Based Language. *Lisp and Symbolic Computation, Kluwer*, 9(2/3):153–179, May/June 1996.
21. J. McAffer. Meta-level programming with coda. In *Proceedings of ECOOP'95*, number 952 in Lecture Notes in Computer Science, pages 190–214. AITO, Springer-Verlag, 1995.
22. H. Okamura, Y. Ishikawa, and M. Tokoro. Al-1/d: A distributed programming system with multi-model reflection framework. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architectures, Reflection and Metalevel Architectures*, pages 36–47. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.
23. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, 1992. Tech. Report UIUCDCS-R-92-1759.
24. Projtech-Technology. Executable UML, http://www.projtech.com/pubs/xuml.html.
25. Softeam. UML Profiles and the J language: Totally control your application development using UML. In *http://www.softeam.fr/us/pdf/uml_profiles.pdf*, 1999.
26. G. Sunyé, A. Le Guennec, and J.-M. Jézéquel. Design pattern application in UML. In E. Bertino, editor, *ECOOP'2000 proceedings*, number 1850, pages 44–62. Lecture Notes in Computer Science, Springer Verlag, June 2000.
27. S. Tan, D. Raila, and R. Campbell. An Object-Oriented Nano-Kernel for Operating System Hardware Support. In *Proceedings of the International Workshop*

on Object-Orientation in Operating Systems, IWOOOS'95. IEEE, Coputer Society Press, 1995.
28. P. Tarr, H. Ossher, and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE'99 Los Angeles CA*, 1999.